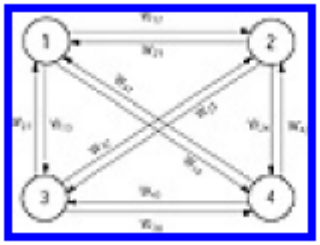
Konstruowanie sieci neuronowej

Pierwszy przykład implementacji C++

Sieć neuronowa, którą przedstawiliśmy w Części 1, jest przykładem sieci Hopfielda z pojedynczą warstwą. Teraz przedstawiamy implementację tej sieci w C++. Załóżmy, że umieściliśmy na tej warstwie cztery neurony, wszystkie połączone ze sobą, jak pokazano na rysunku .Niektóre z tych połączeń mają wagę dodatnią, a reszta ujemną. Możesz przypomnieć sobie z wcześniejszej prezentacji tego przykładu, że do wyznaczenia macierzy wag użyliśmy dwóch wzorców wejściowych. Sieć przywołuje je, gdy dane wejściowe są prezentowane w sieci, pojedynczo. Wejścia te są binarne i ortogonalne, dzięki czemu zapewnione jest ich stabilne przywoływanie. Każdy składnik binarnego wzorca wejściowego ma wartość 0 lub 1. Dwa wektory są ortogonalne, gdy ich iloczyn skalarny - suma iloczynów odpowiadających im składników - wynosi zero. Przykładem binarnego wzorca wejściowego jest 1 0 1 0 0. Przykładem pary wektorów ortogonalnych jest (0, 1, 0, 0, 1) i (1, 0, 0, 1, 0). Przykładem pary wektorów, które nie są ortogonalne, są (0, 1, 0, 0, 1) i (1, 1, 0, 1, 0). Te dwa ostatnie wektory mają iloczyn skalarny równy 1, różny od 0.



Dwa wzorce, dla których chcemy, aby sieć miała stabilne przywoływanie, to A = (1, 0, 1, 0) i B = (0, 1, 0, 1). Macierz wag W jest podawana w następujący sposób:

$$W = \begin{matrix} 0 & -3 & 3 & -3 \\ -3 & 0 & -3 & 3 \\ 3 & -3 & 0 & -3 \\ -3 & 3 & -3 & 0 \end{matrix}$$

UWAGA: Pozytywne powiązania (wartości ze znakami dodatnimi) zwykle zachęcają do porozumienia w stabilnej konfiguracji, podczas gdy ujemne powiązania (wartości ze znakami ujemnymi) mają tendencję do zniechęcania do porozumienia w stabilnej konfiguracji. Potrzebujemy również funkcji progowej i definiujemy ją za pomocą wartości progowej [theta] w następujący sposób:

$$f(t) = \begin{cases} 1 & \text{if } t >= [\text{theta}] \\ 0 & \text{if } t < [\text{theta}] \end{cases}$$

Wartość progowa [theta] jest wykorzystywana jako wartość odcięcia dla aktywacji neuronu, aby umożliwić mu odpalenie. Aktywacja powinna być równa lub przekraczać wartość progową dla neuronu do odpalenia, co oznacza, że ma wyjście 1. W naszej sieci Hopfielda [theta] jest przyjmowane jako 0. W jedynej warstwie tej sieci znajdują się cztery neurony. Dane wyjściowe pierwszego węzła to dane wyjściowe funkcji progowej. Argumentem dla funkcji progowej jest aktywacja węzła. A aktywacja

węzła jest iloczynem skalarnym wektora wejściowego i pierwszej kolumny macierzy wag. Więc jeśli wektor wejściowy to A, iloczyn skalarny wynosi 3, a f(3) = 1. Iloczyny skalarne drugiego, trzeciego i czwartego węzła wynoszą odpowiednio –6, 3 i –6. Odpowiednimi wyjściami są zatem 0, 1 i 0. Oznacza to, że wyjściem sieci jest wektor (1, 0, 1, 0), który jest taki sam jak wzorzec wejściowy. Dlatego sieć przywołała przedstawiony schemat. W przypadku prezentacji B iloczyn skalarny uzyskany w pierwszym węźle wynosi –6, a wynik wynosi 0. Aktywacje wszystkich czterech węzłów wraz z funkcją progową dają (0, 1, 0, 1) jako wynik z sieci, co oznacza, że sieć przywołała również B. Macierz wag działała dobrze z obydwoma wzorcami wejściowymi i nie musimy jej modyfikować.

Klasy w implementacji C++

W naszej implementacji tej sieci w C++ występują następujące klasy: klasa sieci i klasa neuronu. W naszej implementacji tworzymy sieć z czterema neuronami, a te cztery neurony są ze sobą połączone. Neuron nie jest jednak połączony z samym sobą. Oznacza to, że w skierowanym grafie reprezentującym sieć nie ma krawędzi, w której krawędź jest od jednego węzła do samej siebie. Ale dla uproszczenia moglibyśmy udawać, że takie połączenie istnieje i ma wagę 0, więc macierz wag ma zera na swojej głównej przekątnej. Funkcje, które decydują o aktywacji neuronów i wyjściu z sieci, są deklarowane jako publiczne. Dzięki temu są widoczne i dostępne bez ograniczeń. Aktywacje neuronów są obliczane za pomocą funkcji zdefiniowanych w klasie neuronów. Gdy w sieci neuronowej jest więcej niż jedna warstwa, wyjścia neuronów w jednej warstwie stają się wejściami dla neuronów w następnej warstwie. Aby ułatwić przekazywanie danych wyjściowych z jednej warstwy jako danych wejściowych do innej warstwy, nasze implementacje C++ obliczają wyjścia neuronów w klasie sieci. Z tego powodu funkcja progu jest członkiem klasy sieci. Robimy to również dla sieci Hopfield. Aby sprawdzić, czy sieć uzyskała prawidłowe przywołanie, dokonujesz porównań między prezentowanym wzorcem a wyjściem sieci, komponent po komponencie.

Program C++ dla sieci Hopfield

Dla wygody każdy program C++ składa się z dwóch komponentów: Jednym z nich jest plik nagłówkowy ze wszystkimi deklaracjami klas i listami plików bibliotek dołączanych; drugi to plik źródłowy, który zawiera plik nagłówkowy i szczegółowe opisy funkcji składowych klas zadeklarowanych w pliku nagłówkowym. Umieszczasz również funkcję main w pliku źródłowym. Większość obliczeń jest wykonywana przez funkcje składowe klasy, gdy obiekty klasy są tworzone w funkcji main i wykonywane są wywołania odpowiednich funkcji. Jak wiadomo, plik nagłówkowy ma rozszerzenie .h (lub .hpp), a plik źródłowy ma rozszerzenie .cpp, co wskazuje, że jest to plik kodu C++. Można mieć zawartość pliku nagłówkowego zapisaną na początku pliku .cpp i pracować tylko z jednym plikiem, ale rozdzielenie deklaracji i implementacji na dwa pliki pozwala na zmianę implementacji klasy (.cpp) bez zmiana interfejsu na klasę (.h).

Plik nagłówkowy programu C++ dla sieci Hopfield

Listing 1 zawiera Hop.h, plik nagłówkowy programu C++ dla sieci Hopfield. Wymienione w nim pliki dołączane to stdio.h, iostream.h i math.h. Plik iostream.h zawiera deklaracje i szczegóły strumieni C++ dla danych wejściowych i wyjściowych. Klasa sieci i klasa neuronów są zadeklarowane w Hop.h. Elementy członkowskie danych i funkcje członkowskie są deklarowane w każdej klasie, a ich dostępność jest określana za pomocą słów kluczowych protected lub public.

Listing 1Plik nagłówkwy dla programu C++ dla sieci Hopfield.

```
//Hop.h V. Rao, H. Rao

//Single layer Hopfield Network with 4 neurons
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
class neuron
{ protected:
int activation;
friend class network;
public:
int weightv[4];
neuron() {};
neuron(int *j) ;
int act(int, int*);
};
class network
{
public:
neuron nrn[4];
int output[4];
int threshld(int) ;
void activation(int j[4]);
network(int*,int*,int*,int*);
};
```

Uwagi dotyczące pliku nagłówkowego Hop.h

Zauważ, że aktywacja elementu danych w klasie neuronu jest zadeklarowana jako chroniona. Aby aktywacja członków klasy neuronu była dostępna dla klasy sieci, sieć jest zadeklarowana jako klasa zaprzyjaźniona w klasie neuron. Ponadto istnieją dwa konstruktory dla neuronu klasy. Jeden z nich tworzy neuron obiektowy bez inicjowania żadnych składowych danych. Drugi tworzy neuron obiektowy i inicjuje wagi połączeń.

Kod źródłowy sieci Hopfield

Listing 2 zawiera kod źródłowy programu C++ dla sieci Hopfield w pliku Hop.cpp. W tym miejscu zaimplementowano funkcje składowe klas zadeklarowanych w Hop.h. Funkcja main zawiera wzorce wejściowe, wartości inicjujące macierz wag oraz wywołania konstruktora klasy sieciowej i innych funkcji składowych klasy sieciowej.

Listing 2 Kod źródłowy programu C++ dla sieci Hopfield.

```
//Hop.cpp V. Rao, H. Rao
//Single layer Hopfield Network with 4 neurons
#include "hop.h"
neuron::neuron(int *j)
{
int i;
for(i=0;i<4;i++)
{
weightv[i]= *(j+i);
}
}
int neuron::act(int m, int *x)
{
int i;
int a=0;
for(i=0;i<m;i++)
{
a += x[i]*weightv[i];
}
return a;
}
int network::threshld(int k)
{
if(k>=0)
return (1);
else
return (0);
}
network::network(int a[4],int b[4],int c[4],int d[4])
{
nrn[0] = neuron(a) ;
```

```
nrn[1] = neuron(b) ;
nrn[2] = neuron(c) ;
nrn[3] = neuron(d) ;
}
void network::activation(int *patrn)
{
int i,j;
for(i=0;i<4;i++)
{
for(j=0;j<4;j++)
{
cout<<"\n nrn["<<i<<"].weightv["<<j<<"] is "
<<nrn[i].weightv[j];
}
nrn[i].activation = nrn[i].act(4,patrn);
cout<<"\nactivation is "<<nrn[i].activation;
output[i]=threshld(nrn[i].activation);
cout<<"\noutput value is "<<output[i]<<"\n";
}
}
void main ()
{
int patrn1[]= {1,0,1,0},i;
int wt1[]= {0,-3,3,-3};
int wt2[]= {-3,0,-3,3};
int wt3[]= {3,-3,0,-3};
int wt4[]= {-3,3,-3,0};
cout<<"\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF";
cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE";
cout<<"\nPATTERNS 1010 AND 0101 CORRECTLY.\n";
//create the network by calling its constructor.
```

```
// the constructor calls neuron constructor as many times as the number of
// neurons in the network.
network h1(wt1,wt2,wt3,wt4);
//present a pattern to the network and get the activations of the neurons
h1.activation(patrn1);
//check if the pattern given is correctly recalled and give message
for(i=0;i<4;i++)
{
if (h1.output[i] == patrn1[i])
cout<<"\n pattern= "<<patrn1[i]<<
" output = "<<h1.output[i]<<" component matches";
else
cout<<"\n pattern= "<<patrn1[i]<<
" output = "<<h1.output[i]<<
" discrepancy occurred";
}
cout<<"\n\n";
int patrn2[]= {0,1,0,1};
h1.activation(patrn2);
for(i=0;i<4;i++)
{
if (h1.output[i] == patrn2[i])
cout<<"\n pattern= "<<patrn2[i]<<
" output = "<<h1.output[i]<<" component matches";
else
cout<<"\n pattern= "<<patrn2[i]<<
" output = "<<h1.output[i]<<
" discrepancy occurred";
}
}
```

Komentarze do programu C++ dla SieciHopfield

Zwróć uwagę na użycie operatora strumienia wyjściowego cout<< do wyprowadzania tekstu jak ciągi znaków lub wyjście numeryczne. C++ ma klasy istream i ostream, z których pochodzi klasa iostream. Standardowymi strumieniami wejściowymi i wyjściowymi są odpowiednio cin i cout, używane odpowiednio z operatorami >> i <<. Użycie cout dla strumienia wyjściowego jest znacznie prostsze niż użycie funkcji C printf. Jak widać, nie ma sugerowanego formatowania danych wyjściowych. Istnieje jednak przepis, który pozwala sformatować wyjście podczas korzystania z cout. Zwróć także uwagę na sposób wprowadzania komentarzy w programie. Linia z komentarzami powinna zaczynać się podwójnym ukośnikiem //. W przeciwieństwie do C, komentarz nie musi kończyć się podwójnym ukośnikiem. Oczywiście, jeśli komentarze rozciągają się na kolejne wiersze, każdy taki wiersz powinien mieć na początku podwójny ukośnik. Nadal możesz używać pary, /* na początku z */ na końcu linii komentarzy, tak jak to robisz w C. Jeśli komentarz będzie kontynuowany przez wiele linii, łatwiej będzie rozgraniczyć komentarze za pomocą funkcji C. Neurony w sieci należą do klasy sieci i są identyfikowane skrótem nrn. W programie dwa wzorce, 1010 i 0101, są prezentowane w sieci pojedynczo.

Dane wyjściowe z programu C++ dla sieci Hopfield

Wynik tego programu jest następujący i nie wymaga wyjaśnień. Po uruchomieniu tego programu prawdopodobnie zobaczysz wiele danych wyjściowych, więc aby spokojnie spojrzeć na dane wyjściowe, użyj przekierowania. Wpisz Hop > nazwa pliku, a wynik zostanie zapisany w pliku, który możesz edytować za pomocą dowolnego edytora tekstu lub listy, używając typu polecenia filename | more .

THIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF 4 FULLY

INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE PATTERNS 1010 AND

0101 CORRECTLY.

nrn[0].weightv[0] is 0

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is 3

output value is 1

nrn[1].weightv[0] is -3

nrn[1].weightv[1] is 0

nrn[1].weightv[2] is -3

nrn[1].weightv[3] is 3

activation is -6

output value is 0

nrn[2].weightv[0] is 3

nrn[2].weightv[1] is -3

nrn[2].weightv[2] is 0

nrn[2].weightv[3] is -3

activation is 3

output value is 1

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[2] is -3

nrn[3].weightv[3] is 0

activation is -6

output value is 0

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

nrn[0].weightv[0] is 0

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is -6

output value is 0

nrn[1].weightv[0] is -3

nrn[1].weightv[1] is 0

nrn[1].weightv[2] is -3

nrn[1].weightv[3] is 3

activation is 3

output value is 1

nrn[2].weightv[0] is 3

nrn[2].weightv[1] is -3

nrn[2].weightv[2] is 0

nrn[2].weightv[3] is -3

activation is -6

output value is 0

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[2] is -3

nrn[3].weightv[3] is 0

activation is 3

output value is 1

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

Dalsze uwagi na temat programu i jego wyników

Przypomnijmy nasze wcześniejsze omówienie tego przykładu w Części 1. Co sieć daje jako wynik, jeśli przedstawimy wzorzec inny niż zarówno A, jak i B? Jeśli C = (0, 1, 0, 0) jest wzorcem wejściowym, aktywacja (iloczyny skalarne) wyniesie –3, 0, –3, 3, co daje wyjścia (stan następny) neuronów 0,1,0, 1, aby B został odwołany. Jest to dość interesujące, ponieważ gdybyśmy chcieli wprowadzić B, a popełniliśmy drobny błąd i zamiast tego przedstawiliśmy C, sieć wywoła B. Możesz uruchomić program, zmieniając wzorzec na 0, 1, 0, 0 i kompilacja ponownie, aby zobaczyć, że wzorzec B został przywołany. Innym elementem przykładu z rozdziału 1 jest to, że macierz wag W nie jest jedyną macierzą wag, która umożliwiłaby sieci poprawne przywołanie wzorców A i B. Jeśli zastąpimy 3 i –3 w macierzy odpowiednio 2 i –2, otrzymana macierz ułatwi tę samą wydajność sieci. Jednym ze sposobów sprawdzenia tego jest odpowiednia zmiana wt1, wt2, wt3, wt4 podanych w programie oraz skompilowanie i ponowne uruchomienie programu. Powodem, dla którego działają obie matryce wagowe, jest to, że są ze sobą ściśle powiązane. W rzeczywistości jeden jest skalarną (stałą) wielokrotnością drugiego, to znaczy, jeśli pomnożysz każdy element macierzy przez ten sam skalar, czyli 2/3, otrzymasz odpowiednią macierz w przypadku zastąpienia 3 i -3 odpowiednio z 2 i –2.

Nowa macierz wag, która przywołuje więcej wzorców

Kontynuujmy omawianie tego przykładu. Załóżmy, że jesteśmy zainteresowani tym, aby oprócz wzorców A i B również prawidłowo przywołać wzorce E = (1, 0, 0, 1) i F = (0, 1, 1, 0). Będziemy trenować sieć i wymyślić algorytm uczenia się, który omówimy bardziej szczegółowo w dalszej części . Wymyślamy macierz W_1, która następuje.

$$W_1 = \begin{matrix} 0 & -5 & 4 & 4 \\ -5 & 0 & 4 & 4 \\ 4 & 4 & 0 & -5 \\ 4 & 4 & -5 & 0 \end{matrix}$$

Spróbuj użyć tej modyfikacji macierzy wag w programie źródłowym, a następnie skompiluj i uruchom program, aby zobaczyć, że sieć pomyślnie przywołuje wszystkie cztery wzorce A, B, E i F.

UWAGA: Przedstawiona implementacja C++ nie zawiera funkcji aktualizacji asynchronicznej wspomnianej w Rozdziale 1, która nie jest konieczna dla przedstawionych wzorców. Kodowanie tej funkcji pozostawiono jako ćwiczenie dla czytelnika.

Oznaczanie wagi

Być może zastanawiasz się, w jaki sposób te macierze wag zostały opracowane w poprzednim przykładzie, ponieważ do tej pory omawialiśmy tylko, jak sieć wykonuje swoją pracę i jak wdrożyć model. Nauczyłeś się, że wybór matrycy wagowej niekoniecznie jest wyjątkowy. Ale chcesz mieć pewność, że istnieje pewien ustalony sposób, poza próbami i błędami, na skonstruowanie macierzy wag. Możesz przejść do tego w następujący sposób.

Mapowanie binarne na bipolarne

Spójrzmy na poprzedni przykład. Widziałeś, że zastępując każde 0 w ciągu binarnym przez –1, otrzymujesz odpowiedni ciąg dwubiegunowy. Jeśli zachowasz wszystkie jedynki i zamienisz każde 0 na –1, otrzymasz formułę dla powyższej opcji. Możesz zastosować następującą funkcję do każdego bitu w ciągu:

$f(x) = 2x – 1$

UWAGA: Kiedy podasz binarny bit x, otrzymasz odpowiedni dwubiegunowy znak f(x)

W przypadku mapowania odwrotnego, które zamienia ciąg bipolarny w ciąg binarny, użyj następującej funkcji:

$f(x) = (x + 1) / 2$

UWAGA: Kiedy podasz dwubiegunowy znak x, otrzymasz odpowiedni bit binarny f(x)

Wkład wzorca do wagi

Następnie pracujemy z bipolarnymi wersjami wzorców wejściowych. Bierzesz każdy wzorzec do przywołania, pojedynczo, i określasz jego udział w macierzy wag sieci. Wkład każdego wzoru sam w sobie jest macierzą. Wielkość takiej macierzy jest taka sama jak macierz wag sieci. Następnie dodaj te wkłady w sposób, w jaki dodawane są macierze, a otrzymasz macierz wag dla sieci, która jest również nazywana macierzą korelacji. Znajdźmy wkład wzorca A = (1, 0, 1, 0): Po pierwsze, zauważymy, że odwzorowanie binarne na bipolarne A = (1, 0, 1, 0) daje wektor (1, –1 , 1, –1). Następnie bierzemy transpozycję i mnożymy, tak jak macierze są mnożone i widzimy co następuje:

$$\begin{matrix}1\\1\\1\\1\end{matrix}\begin{bmatrix}1 & -1 & 1 & -1\end{bmatrix} = \begin{bmatrix}1 & -1 & 1 & -1\\ -1 & 1 & -1 & 1\\ 1 & -1 & 1 & -1\\ -1 & 1 & -1 & 1\end{bmatrix}$$

Teraz odejmij 1 od każdego elementu na głównej przekątnej (która biegnie od lewego górnego rogu do prawego dolnego). Ta operacja daje taki sam wynik, jak odjęcie macierzy jednostkowej od podanej macierzy i otrzymanie zer na głównej przekątnej. Otrzymana macierz, która jest podana dalej, jest wkładem wzorca (1, 0, 1, 0) do macierzy wag.

$$\begin{matrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{matrix}$$

Podobnie, możemy obliczyć wkład z wzorca B = (0, 1, 0, 1), sprawdzając, czy wkład wzorca B jest tą samą macierzą, co wkład wzorca A. Dlatego macierz wag dla tego ćwiczenia jest macierzą W pokazaną tutaj.

$$W = \begin{matrix} 0 & -2 & 2 & -2 \\ -2 & 0 & -2 & 2 \\ 2 & -2 & 0 & -2 \\ -2 & 2 & -2 & 0 \end{matrix}$$

Możesz teraz opcjonalnie zastosować dowolny mnożnik skalarny do wszystkich wpisów macierzy, jeśli chcesz. W ten sposób wcześniej uzyskaliśmy wartości +/- 3 zamiast pokazanych powyżej +/- 2 wartości.

Sieć autoasocjacyjna

Pokazana właśnie sieć Hopfield ma cechę polegającą na tym, że sieć kojarzy ze sobą wzorzec wejściowy podczas przywołania. To sprawia, że sieć staje się siecią autoasocjacyjną. Wzorce używane do wyznaczania właściwej macierzy wag są również tymi, które są autoasocjacyjnie przywołany. Te wzorce nazywane są wzorami. Wzorzec inny niż przykładowy może, ale nie musi być przywołany przez sieć. Oczywiście, gdy przedstawiasz wzorzec 0 0 0 0, jest on stabilny, mimo że nie jest wzorcowym wzorcem.

Ortogonalne wzory bitów

Być może zastanawiasz się, ile wzorców jest w stanie przywołać sieć z czterema węzłami. Rozważmy najpierw, ile różnych wzorców bitowych jest ortogonalnych do danego wzorca bitowego. To pytanie tak naprawdę odnosi się do wzorców bitowych, w których przynajmniej jeden bit jest równy 1. Trochę odbicie mówi nam, że jeśli dwa wzorce bitowe mają być ortogonalne, to oba nie mogą mieć jedynek w tej samej pozycji, ponieważ iloczyn skalarny musiałby wynosić 0. Innymi słowy, musi skutkować bitową operacją logiczną AND dwóch wzorców bitowych w 0. To sugeruje, co następuje. Jeśli wzorzec P ma k, mniej niż 4, pozycje bitowe z 0 (a więc pozycje 4-k bitowe z 1) i jeśli wzorzec Q ma być ortogonalny do P, to Q może mieć 0 lub 1 w tych k pozycjach, ale musi mieć tylko 0 w pozostałych 4-k pozycjach. Ponieważ istnieją dwie możliwości wyboru dla każdej z pozycji k, istnieje 2k możliwych wzorów prostopadłych do P. Ta liczba 2^k wzorów obejmuje wzór ze wszystkimi zerami. Tak więc naprawdę istnieją 2^k-1 niezerowe wzory ortogonalne do P. Niektóre z tych 2k-1 wzorów nie są do siebie ortogonalne. Na przykład P może być wzorcem 0 1 0 0, który ma k = 3 pozycje z 0. Istnieje 2^3–1=7 niezerowych wzorców ortogonalnych do 0 1 0 0. Wśród nich są wzorce 1 0 1 0 i 1 0 0 1, które nie są do siebie prostopadłe, ponieważ ich iloczyn skalarny wynosi 1, a nie 0.

Węzły sieciowe i wzorce wejściowe

Ponieważ nasza sieć ma w sobie cztery neurony, ma również cztery węzły na grafie skierowanym, który reprezentuje sieć. Są one połączone bocznie, ponieważ połączenia są ustanawiane od węzła do węzła. Są boczne, ponieważ wszystkie węzły znajdują się w tej samej warstwie. Zaczęliśmy od wzorców A = (1, 0, 1, 0) i B = (0, 1, 0, 1) jako przykładów. Jeśli weźmiemy jakikolwiek inny niezerowy wzorzec, który jest ortogonalny do A, będzie miał 1 w pozycji, w której B również ma 1. Więc nowy wzorzec nie będzie

ortogonalny do B. Dlatego ortogonalny zestaw wzorców, który zawiera A i B może mieć tylko te dwa elementy jako swoje elementy. Jeśli usuniesz B ze zbioru, możesz uzyskać (co najwyżej) dwie inne osoby, które połączą się z A, tworząc zbiór ortogonalny. Są to wzorce (0, 1, 0, 0) i (0, 0, 0, 1). Jeśli wykonasz opisaną wcześniej procedurę, aby uzyskać macierz korelacji, otrzymasz następującą macierz wag:

$$W = \begin{matrix} 0 & -1 & 3 & -1 \\ -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{matrix}$$

W przypadku tej macierzy przywoływany jest wzorzec A, ale wzorzec zerowy (0, 0, 0, 0) jest uzyskiwany dla dwóch wzorców (0, 1, 0, 0) i (0, 0, 0, 1). Po uzyskaniu wzorca zerowego jego własne przywołanie będzie stabilne.

Drugi przykład implementacji C++

Przypomnijmy grę kasową z serialu The Price is Right, użytego jako jeden z przykładów w Części 1. Ten przykład doprowadził do opisu sieci neuronowej Perceptron. Wrócimy teraz do naszej dyskusji na temat modelu Perceptron i kontynuujemy jego implementację w C++. Miej na uwadze przykład gry kasowej, czytając następującą implementację C++ modelu Perceptron. Należy również zauważyć, że sygnały wejściowe w tym przykładzie niekoniecznie są binarne, ale mogą być liczbami rzeczywistymi. Dzieje się tak dlatego, że ceny przedmiotów, które zawodnik musi wybrać, są liczbami rzeczywistymi (dolary i centy). Perceptron ma jedną warstwę neuronów wejściowych i jedną warstwę neuronów wyjściowych. Każdy neuron warstwy wejściowej jest połączony z każdym neuronem w warstwie wyjściowej.

Implementacja C++ sieci Perceptron

W naszej implementacji tej sieci w C++ mamy klasy: mamy osobne klasy dla neuronów wejściowych i wyjściowych. Klasa ineuron jest przeznaczona dla neuronów wejściowych. Ta klasa ma wagę i aktywację jako składowe danych. Klasa neuronu jest podobna i dotyczy neuronu wyjściowego. Jest deklarowany jako klasa przyjacielska w klasie ineuron. Klasa neuronu wyjściowego ma również element danych zwany wyjściem. Istnieje klasa sieci, która jest klasą przyjaciela w klasie oneuron. Instancja klasy network jest tworzona z czterema neuronami wejściowymi. Te cztery neurony są połączone jednym neuronem wyjściowym. Funkcje składowe klasy ineuron to: (1) domyślny konstruktor, (2) drugi konstruktor, który jako argument przyjmuje liczbę rzeczywistą oraz (3) funkcja obliczająca wyjście neuronu wejściowego. Konstruktor przyjmujący jeden argument używa tego argumentu do ustawienia wartości wagi połączenia między neuronem wejściowym a wyjściowym. Funkcje, które decydują o aktywacji neuronów i wyjściu z sieci, są deklarowane jako publiczne. Aktywacje neuronów są obliczane za pomocą funkcji zdefiniowanych w klasach neuronów. Wartość progowa jest używana przez funkcję składową neuronu wyjściowego do określenia, czy aktywacja neuronu jest wystarczająco duża, aby mógł się uruchomić, dając wynik równy 1.

Plik nagłówka

Listing 3 zawiera percept.h, plik nagłówkowy programu C++ dla sieci Perceptron. percept.h zawiera deklaracje dla trzech klas, jednej dla neuronów wejściowych, jednej dla neuronów wyjściowych i jednej dla sieci.

Listing 3 percept.h plik nagłówkowy.

```
//percept.h V. Rao, H. Rao
// Perceptron model
#include <stdio.h>
#include <iostream.h>
#include <math.h>
class ineuron
{ protected:
float weight;
float activation;
friend class oneuron;
public:
ineuron() {};
ineuron(float j) ;
float act(float x);
};
class oneuron
{
protected:
int output;
float activation;
friend class network;
public:
oneuron() { };
void actvtion(float x[4], ineuron *nrn);
int outvalue(float j) ;
};
class network
{
public:
ineuron nrn[4];
```

```
oneuron onrn;

network(float,float,float,float);

};
```

Implementacja funkcji

Sieć jest zaprojektowana tak, aby mieć cztery neurony w warstwie wejściowej. Każda z nich jest obiektem klasy ineuron, a są to klasy składowe w sieci klas. Istnieje jeden wyraźnie zdefiniowany neuron wyjściowy klasy oneneuron. Konstruktor sieci wywołuje również konstruktor neuronu dla każdego neuronu warstwy wejściowej w sieci, podając mu początkową wagę połączenia z neuronem w warstwie wyjściowej. Konstruktor dla neuronu wyjściowego jest również wywoływany przez konstruktor sieci, w tym samym czasie inicjując elementy danych wyjściowych i aktywacji neuronu wyjściowego na zero. Aby upewnić się, że istnieje dostęp do potrzebnych informacji i funkcji, neuron wyjściowy jest zadeklarowany jako klasa zaprzyjaźniona w klasie ineuron. Sieć jest zadeklarowana jako klasa przyjacielska w klasie oneneuron.

Kod źródłowy dla sieci Perceptron

Listing 4 zawiera kod źródłowy w percept.cpp dla implementacji C++ modelu Perceptron omówionego wcześniej.

Listing 4 Kod źródłowy modelu Perceptron.

```
//percept.cpp V. Rao, H. Rao

//Perceptron model

#include "percept.h"

#include "stdio.h"

#include "stdlib.h"

ineuron::ineuron(float j)

{

weight= j;

}

float ineuron::act(float x)

{

float a;

a = x*weight;

return a;

}

void oneuron::actvtion(float *inputv, ineuron *nrn)

{
```

```
int i;
activation = 0;
for(i=0;i<4;i++)
{
cout<<"\nweight for neuron "<<i+1<<" is "<<nrn[i].weight;
nrn[i].activation = nrn[i].act(inputv[i]);
cout<<" activation is "<<nrn[i].activation;
activation += nrn[i].activation;
}
cout<<"\n\nactivation is "<<activation<<"\n";
}
int oneuron::outvalue(float j)
{
if(activation>=j)
{
cout<<"\nthe output neuron activation \
exceeds the threshold value of "<<j<<"\n";
output = 1;
}
else
{
cout<<"\nthe output neuron activation \
is smaller than the threshold value of "<<j<<"\n";
output = 0;
}
cout<<" output value is "<< output;
return (output);
}
network::network(float a,float b,float c,float d)
{
nrn[0] = ineuron(a) ;
```

```
nrn[1] = ineuron(b) ;
nrn[2] = ineuron(c) ;
nrn[3] = ineuron(d) ;
onrn = oneuron();
onrn.activation = 0;
onrn.output = 0;
}
void main (int argc, char * argv[])
{
float inputv1[]= {1.95,0.27,0.69,1.25};
float wtv1[]= {2,3,3,2}, wtv2[]= {3,0,6,2};
FILE * wfile, * infile;
int num=0, vecnum=0, i;
float threshold = 7.0;
if (argc < 2)
{
cerr << "Usage: percept Weightfile Inputfile";
exit(1);
}
// open files
wfile= fopen(argv[1], "r");
infile= fopen(argv[2], "r");
if ((wfile == NULL) || (infile == NULL))
{
cout << " Can't open a file\n";
exit(1);
}
cout<<"\nTHIS PROGRAM IS FOR A PERCEPTRON NETWORK WITH AN INPUT LAYER OF";
cout<<"\n4 NEURONS, EACH CONNECTED TO THE OUTPUT NEURON.\n";
cout<<"\nTHIS EXAMPLE TAKES REAL NUMBERS AS INPUT SIGNALS\n";
//create the network by calling its constructor.
```

```
//the constructor calls neuron constructor as many times as the number of

//neurons in input layer of the network.

cout<<"please enter the number of weights/vectors \n";

cin >> vecnum;

for (i=1;i<=vecnum;i++)

{

fscanf(wfile,"%f %f %f %f\n", &wtv1[0],&wtv1[1],&wtv1[2],&wtv1[3]);

network h1(wtv1[0],wtv1[1],wtv1[2],wtv1[3]);

fscanf(infile,"%f %f %f %f \n",

&inputv1[0],&inputv1[1],&inputv1[2],&inputv1[3]);

cout<<"this is vector # " << i << "\n";

cout << "please enter a threshold value, eg 7.0\n";

cin >> threshold;

h1.onrn.actvtion(inputv1, h1.nrn);

h1.onrn.outvalue(threshold);

cout<<"\n\n";

}

fclose(wfile);

fclose(infile);

}
```

Komentarze do Twojego programu C++

Zwróć uwagę na użycie operatora strumienia wejściowego cin>> w programie C++ zamiast funkcji C scanf w kilku miejscach. Klasa iostream w C++ została omówiona wcześniej w tym rozdziale. Program działa tak : Najpierw neurony wejściowe sieci otrzymują wagi połączeń, a następnie wektor wejściowy jest prezentowany warstwie wejściowej. Określona jest wartość progowa, a neuron wyjściowy wykonuje ważoną sumę swoich wejść, które są wyjściami neuronów warstwy wejściowej. Ta ważona suma jest aktywacją neuronu wyjściowego i jest porównywana z wartością progową, a neuron wyjściowy odpala (wyjście wynosi 1), jeśli wartość progowa nie jest większa niż jego aktywacja. Nie odpala (wyjście wynosi 0), jeśli jego aktywacja jest mniejsza od wartości progowej. W tym wdrożeniu nie uwzględnia się szkoleń nadzorowanych ani nienadzorowanych.

Wejście/wyjście dla percept.cpp

W tym programie używane są dwa pliki danych. Jeden służy do ustawiania wag, a drugi do ustawiania wektorów wejściowych. W wierszu poleceń wpisujesz nazwę programu, a następnie wagę z nazwą pliku i nazwę pliku wejściowego. Do tej dyskusji (również na dysku towarzyszącym tej książce) utwórz plik o nazwie weight.dat, który zawiera następujące dane:

```
2.0 3.0 3.0 2.0
3.0 0.0 6.0 2.0
```

To są dwa wektory wagowe. Utwórz również plik wejściowy o nazwie input.dat z dwoma wektorami danych poniżej:

```
1.95 0.27 0.69 1.25
0.30 1.05 0.75 0.19
```

Podczas wykonywania programu najpierw zostaniesz zapytany o liczbę używanych wektorów (w tym przypadku 2), a następnie o wartość progową dla wektorów danych wejściowych/wagi (w obu przypadkach użyj 7,0). Zobaczysz wtedy następujące dane wyjściowe. Zwróć uwagę, że dane wprowadzane przez użytkownika są zapisane kursywą.

percept weight.dat input.dat

THIS PROGRAM IS FOR A PERCEPTRON NETWORK WITH AN INPUT LAYER OF 4

NEURONS, EACH CONNECTED TO THE OUTPUT NEURON.

THIS EXAMPLE TAKES REAL NUMBERS AS INPUT SIGNALS

please enter the number of weights/vectors

2

this is vector # 1

please enter a threshold value, eg 7.0

7.0

weight for neuron 1 is 2 activation is 3.9

weight for neuron 2 is 3 activation is 0.81

weight for neuron 3 is 3 activation is 2.07

weight for neuron 4 is 2 activation is 2.5

activation is 9.28

the output neuron activation exceeds the threshold value of 7

output value is 1

this is vector # 2

please enter a threshold value, eg 7.0

7.0

weight for neuron 1 is 3 activation is 0.9

weight for neuron 2 is 0 activation is 0

weight for neuron 3 is 6 activation is 4.5

weight for neuron 4 is 2 activation is 0.38

activation is 5.78

the output neuron activation is smaller than the threshold value of 7

output value is 0

Na koniec spróbuj dodać wektor danych (1,4, 0,6, 0,35, 0,99) do pliku danych. Dodaj wektor wag (2, 6, 8, 3) do pliku wag i użyj wartości progowej 8,25, aby zobaczyć wynik. Możesz również użyć innych wartości do eksperymentowania.

Modelowanie sieci

Do tej pory rozważaliśmy budowę dwóch sieci, pamięci Hopfielda i Perceptronu. Jakie są inne kwestie (które zostaną omówione bardziej szczegółowo później), o których należy pamiętać? Niektóre z rozważań dotyczących modelowania sieci neuronowej dla aplikacji to:

```
nature of inputs
        fuzzy
                  binary
                  analog
        crisp
                  binary
                  analog
number of inputs
nature of outputs
        fuzzy
                  binary
                  analog
        crisp
                  binary
                  analog

number of outputs

nature of the application
        to complete patterns (recognize corrupted patterns)
        to classify patterns
        to do an optimization
        to do approximation
        to perform data clustering
        to compute functions

dynamics
    adaptive
              learning
                        training
                                  with exemplars
```

```
                                        without exemplars

            self-organizing

    nonadaptive

            learning

                        training

                                        with exemplars
                                        without exemplars

            self-organizing

hidden layers

    number

            fixed
            variable

    sizes

            fixed
            variable

processing

       additive
       multiplicative
       hybrid

                 additive and multiplicative
                 combining other approaches
                             expert systems
                             genetic algorithms
```

Modele hybrydowe, jak wskazano powyżej, mogą być różnorodnym połączeniem podejścia sieci neuronowych z metodami systemów eksperckich lub łączenia paradygmatów przetwarzania addytywnego i multiplikatywnego. Systemy wspomagania decyzji są podatne na podejścia łączące sieci neuronowe z systemami eksperckimi. Przykładem modelu hybrydowego łączącego różne tryby przetwarzania przez neurony jest sieć neuronowa Sigma Pineural, w której jedna warstwa neuronów wykorzystuje sumowanie w agregacji, a kolejna warstwa neuronów wykorzystuje przetwarzanie multiplikatywne. Ukryta warstwa, jeśli tylko jedna, w sieci neuronowej to warstwa neuronów, która działa pomiędzy warstwą wejściową a wyjściową sieci. Neurony w tej warstwie odbierają dane wejściowe od tych w warstwie wejściowej i dostarczają swoje dane wyjściowe jako dane wejściowe do neuronów w warstwie wyjściowej. Gdy ukryta warstwa znajduje się między innymi ukrytymi warstwami, otrzymuje dane wejściowe i dostarcza dane wejściowe do odpowiednich ukrytych warstw. Podczas modelowania sieci często nie jest łatwo określić, ile, jeśli w ogóle, ukrytych warstw i jakich rozmiarów jest potrzebnych w modelu. Niektóre podejścia, takie jak algorytmy genetyczne – które są paradygmatami konkurującymi z podejściami sieci neuronowych w wielu sytuacjach, ale mimo to mogą być kooperatywne, jak tutaj – są czasami wykorzystywane do określenia potrzebnej lub optymalnej, w zależności od przypadku, liczby ukrytych warstwy i/lub neurony w tych ukrytych warstwach. Poniżej przedstawiamy jedną z takich aplikacji.

Ktoś gra w kółko i krzyżyk?

David Fogel opisuje ewolucyjne ogólne rozwiązywanie problemów i używa jako przykładu znanej gry w kółko i krzyżyk. Chodzi o to, aby wymyślić optymalne strategie w tej grze. Znacznikiem pierwszego

gracza jest X, a znacznikiem drugiego gracza jest O. Ktokolwiek zdobędzie trzy swoje znaczniki w rzędzie, kolumnie lub po przekątnej, zanim zrobi to drugi gracz, wygrywa. Sprytni gracze radzą sobie z remisem, jeśli ich równie sprytny przeciwnik udaremni ich próby wygranej. Pozycja remisu to taka, w której żaden z graczy nie ma trzech swoich znaczników w rzędzie, kolumnie lub po przekątnej. Tablicę można opisać za pomocą wektora dziewięciu elementów, z których każdy jest liczbą o trzech wartościach. Wyobraź sobie kwadraty planszy do gry ułożone w kolejności rząd po rzędzie od góry do dołu. Pozwól, aby 1 wskazywało obecność X w tym kwadracie, 0 oznaczało puste miejsce, a -1 odpowiadało O. To jest przykład kodowania statusu planszy. Na przykład (-1, 0, 1, 0, -1, 0, 1, 1, -1) to wygrywająca pozycja dla drugiego gracza, ponieważ odpowiada szachownicy wyglądającej jak poniżej.

O X
 O
X X O

Sieć neuronowa dla tego problemu będzie miała warstwę wejściową z dziewięcioma neuronami, ponieważ każdy wzorzec wejściowy ma dziewięć komponentów. Byłoby kilka ukrytych warstw. Ale przykład dotyczy jednej ukrytej warstwy. Warstwa wyjściowa zawiera również dziewięć neuronów, dzięki czemu jeden cykl działania sieci pokazuje, jaka ma być najlepsza konfiguracja płytki przy danym wejściu. Oczywiście, podczas tego cyklu operacji, wszystko, co należy określić, to które puste miejsce, oznaczone na wejściu zerem, powinno zostać zmienione na 1, jeśli strategia jest stosowana dla gracza 1. Żadna z jedynek i -1 ma zostać zmieniony. W tym konkretnym przykładzie sama architektura sieci neuronowej jest dynamiczna. Sieć rozszerza się lub kurczy według pewnych zasad, które opisano dalej. Fogel opisuje sieć jako sieć ewoluującą w tym sensie, że liczba neuronów w warstwie ukrytej zmieniała się z prawdopodobieństwem 0,5. Węzeł miał takie samo prawdopodobieństwo dodania lub usunięcia. Ponieważ liczba nieoznaczonych kwadratów maleje po każdym zagraniu, takie podejście z różną liczbą neuronów w sieci wydaje się rozsądne i interesujące. Początkowy zestaw wag to wartości losowe od -0,5 do 0,5 włącznie, zgodnie z rozkładem równomiernym. Odchylenie i wartości progowe również pochodzą z tego rozkładu. Funkcja esicy:

$1/(1 + e^{-x})$

służy również do określenia wyjść.

Wagi i błędy systematyczne ulegały zmianie podczas cykli szkolenia w zakresie obsługi sieci. W ten sposób sieć miała fazę uczenia się. Sieć ta jest adaptacyjna, ponieważ zmienia swoją architekturę. Inne formy adaptacji w sieciach neuronowych polegają na zmianie wartości parametrów dla architektury stałej. Wyniki eksperymentu opisanego przez Fogla pokazują, że potrzeba dziewięciu neuronów również w warstwie ukrytej, aby Twoja sieć była najlepsza dla tego problemu. Oczyścili także każdą strategię, która mogła stracić. Fogel kładzie nacisk na ewolucyjny aspekt procesu lub eksperymentu adaptacyjnego. Nasze zainteresowanie tym przykładem wynika przede wszystkim z zastosowania adaptacyjnej sieci neuronowej. Wybór gry w kółko i krzyżyk, będąc prostą i aż nazbyt znajomą grą, należy do gatunku znacznie bardziej skomplikowanych gier. Te gry wymagają od gracza umieszczenia znacznika na jakiejś pozycji w danej tablicy, a gdy gracze na zmianę robią to, pewne kryterium określa, czy jest to remis, czy też kto wygrał. W przeciwieństwie do gry w kółko i krzyżyk, kryterium zwycięstwa może nie być znane graczom.

Stabilność i plastyczność

Omówimy teraz kilka innych kwestii związanych z modelowaniem sieci neuronowych, wprowadzając koncepcje pamięci krótkotrwałej i pamięci długotrwałej. Trening sieci neuronowych jest zwykle wykonywany w sposób iteracyjny, co oznacza, że procedura jest powtarzana określoną liczbę razy.

Te iteracje są nazywane cyklami. Po każdym cyklu zastosowane dane wejściowe mogą pozostać takie same lub ulec zmianie, albo wagi mogą pozostać takie same lub ulec zmianie. Taka zmiana jest oparta na wyniku zakończonego cyklu. Jeżeli liczba cykli nie jest ustawiona, a sieć może przechodzić przez cykle aż do spełnienia jakiegoś innego kryterium, naturalnie pojawia się pytanie, czy w końcu nastąpi zakończenie procesu iteracyjnego.

Stabilność sieci neuronowej

Stabilność odnosi się do takiej zbieżności, która ułatwia zakończenie procesu iteracyjnego. Na przykład, jeśli dowolne dwa kolejne cykle dają w sieci te same dane wyjściowe, może nie być potrzeby wykonywania większej liczby iteracji. W tym przypadku nastąpiła konwergencja, a sieć ustabilizowała się w swoim działaniu. Jeżeli wagi są modyfikowane po każdym cyklu, to zbieżność wag stanowiłaby stabilność sieci. W niektórych sytuacjach potrzeba o wiele więcej iteracji, niż chcesz, aby wyniki w dwóch kolejnych cyklach były takie same. Następnie można zastosować poziom tolerancji na kryterium zbieżności. Przy poziomie tolerancji osiągasz wczesne, ale satysfakcjonujące zakończenie działania sieci.

Plastyczność dla sieci neuronowej

Załóżmy, że sieć jest wytrenowana do uczenia się pewnych wzorców i w tym procesie wagi są dostosowywane zgodnie z algorytmem. Po nauczeniu się tych wzorców i napotkaniu nowego wzorca, sieć może zmodyfikować wagi, aby nauczyć się nowego wzorca. Ale co, jeśli nowa struktura wagowa nie reaguje na nowy wzorzec? Wtedy sieć nie posiada plastyczności - zdolności do zadowalającego radzenia sobie z nową pamięcią krótkotrwałą (STM) przy zachowaniu pamięci długoterminowej (LTM). Próba wyposażenia sieci w plastyczność może mieć niekorzystny wpływ na stabilność sieci.

Pamięć krótkotrwała i pamięć długotrwała

W poprzednim akapicie wspomnieliśmy o pamięci krótkotrwałej (STM) i pamięci długotrwałej (LTM). STM to w zasadzie informacje, które są obecnie i być może tymczasowo przetwarzane. Przejawia się to we wzorcach, które napotyka sieć. Z drugiej strony LTM to informacje, które są już przechowywane i nie są obecnie przetwarzane. W sieci neuronowej STM jest zwykle charakteryzowany przez wzorce, a LTM jest charakteryzowany przez wagi połączeń. Wagi określają, w jaki sposób dane wejściowe są przetwarzane w sieci w celu uzyskania danych wyjściowych. W trakcie cykli działania sieci wagi mogą się zmieniać. Po konwergencji reprezentują LTM, ponieważ osiągnięte poziomy masy są stabilne.

Podsumowanie

Widziałeś implementacje C++ prostej sieci Hopfielda i prostej sieci Perceptron. To, co nie zostało w nich zawarte, to automatyczna iteracja i algorytm uczenia. Nie były one potrzebne, aby przykłady użyte tu pokazały implementację C++, nacisk położono na sposób implementacji. Rozważania dotyczące modelowania sieci neuronowej są przedstawione wraz z zarysem wykorzystania kółka i krzyża jako przykładu adaptacyjnego modelu sieci neuronowej. Zapoznałeś się również z następującymi pojęciami: stabilność, plastyczność, pamięć krótkotrwała i pamięć długotrwała (omówione w dalszych rozdziałach). Można o nich powiedzieć znacznie więcej, jeśli chodzi o tzw. dylemat nasycenia szumem, czyli dylemat stabilność – plastyczność i jakie badania rozwinęły się w celu ich rozwiązania